Appendix A, example XMON source Code

XMON primarily consists of two parts. The first part handles the Debug exception of the Xtensa processor. This is implemented in the file DebugExceptionVectorHandler-mon.S The second part is implemented in xtensa-mon.c and handles the higher level protocol with the debugger.

I. DebugExceptionVectorHandler-mon.S.

```
// Exports
.global _DebugExceptionFromVector
.global _ar_registers
.global _sr_registers
.global _level_0_interrupt
.global _flush_i_cache
.global _xmon_out
.global _xmon_in
.global _xmon_flush
.global _xmon_init

// Imports
//      _handle_exception

#include <machine/specreg.h>
#include "DebugExceptionVectorHandler.h"

#define AR_SAVE_SIZE (4*NUM_AREGS)
#define SR_SAVE_SIZE (4*256)

// Parameters
#define XMON_STACK_SIZE (2048+1024)
```

```
/*************************************************************************

 The assembler portion of the debug handler begins here.  The handler
does three major things.  First, it saves the processor state.  The
bulk of the save sequence saves all the address registers.  Note that
we don't try to save the registers into interrupted process' stack
because it may have become corrupted and the debugger wants to perturb
the processor state as little as possible.  Second, the handler sets
up the run-time environment for the debugger stuff, which we have
written in C.  Third, upon return from the stub, we restore the
interrupted process' registers, and resume the process.  The debugger
can force the process to resume at an alternative pc by overwriting
the saved value of the appropriate EPC.

In the comments below, we will use "ipwb" to refer to the interrupted
process' window base, and "wb" to the current window base.

*************************************************************************/

//      .section .bss
        .section .text
        .align 16
//_ar_registers:
//      .space  AR_SAVE_SIZE
//_sr_registers:
//      .space  SR_SAVE_SIZE
_xmon_stack:
        .space  XMON_STACK_SIZE
_xmon_stack_bot:

        .text
        .begin literal
        .align 4
_xmon_stack_ptr:
        .word   _xmon_stack_bot-4*16
ar_save_ptr:
        .word   _ar_registers
sr_save_area_ptr:
        .word   _sr_registers
//ar_save_area_ptr:
//      .word   _registers+AR0_OFFSET

        .globl _handle_exception
handler:
        .word   _handle_exception

        .align 4
.Laddress_of_save1_table_ptr:
        .word   save1_table_ptr

        .align 4
save1_table_ptr:
        .word   save1_28        /* ipwb=0 */
        .word   save1_24        /* ipwb=1 */
        .word   save1_20        /* ipwb=2 */
        .word   save1_16        /* ipwb=3 */
        .word   save1_12        /* ipwb=4 */
        .word   save1_8 /* ipwb=5 */
        .word   save1_4 /* ipwb=6 */
        .word   save1_0 /* ipwb=7 */

        .align 4
.Laddress_of_save2_table_ptr:
        .word   save2_table_ptr
save2_table_ptr:
        .word   save2_0 /* ipwb=0 */
        .word   save2_4 /* ipwb=1 */
        .word   save2_8 /* ipwb=2 */
        .word   save2_12        /* ipwb=3 */
        .word   save2_16        /* ipwb=4 */
```

```
        .word   save2_20        /* ipwb=5 */
        .word   save2_24        /* ipwb=6 */
        .word   save2_28        /* ipwb=7 */

        .align 4
.LDWOE:
        .word 0xfffbffff
.Lps:
        .word (1<<18)           /* WOE and KM */
.Laddress_of_restore1_table_ptr:
        .word   restore1_table_ptr
restore1_table_ptr:
        .word   restore1_28     /* ipwb=0 */
        .word   restore1_24     /* ipwb=1 */
        .word   restore1_20     /* ipwb=2 */
        .word   restore1_16     /* ipwb=3 */
        .word   restore1_12     /* ipwb=4 */
        .word   restore1_8      /* ipwb=5 */
        .word   restore1_4      /* ipwb=6 */
        .word   restore1_0      /* ipwb=7 */

        .align 4
.Laddress_of_restore2_table_ptr:
        .word   restore2_table_ptr
restore2_table_ptr:
        .word   restore2_0      /* wb=0 */
        .word   restore2_4      /* wb=1 */
        .word   restore2_8      /* wb=2 */
        .word   restore2_12     /* wb=3 */
        .word   restore2_16     /* wb=4 */
        .word   restore2_20     /* wb=5 */
        .word   restore2_24     /* wb=6 */
        .word   restore2_28     /* wb=7 */
        .end literal

        .text
        .align 4

_DebugExceptionFromVector:


        /* Save a0,a1,a2 into various places so that we can setup
           the save sequence.  Notice that we need to take care
           that this code works even when a0, a1 contain the
           same value.  See the NOOP comments.
        */
        l32r    a0,ar_save_ptr
        s32i    a1,a0,4
        s32i    a2,a0,8
        l32r    a2,sr_save_area_ptr
        rsr     a1,WINDOWSTART
        s32i    a1,a2, (WINDOWSTART*4)  /* save windowstart */
        rsr     a1,WINDOWBASE
        s32i    a1,a2,(WINDOWBASE*4)   /* save WB */
        slli    a1,a1,4         /* multiply by 16, size in bytes of
                                   the 4 register window */
        add     a1,a0,a1
        /* At this point:
                a0: address of save area.
                a1: &save_area+wb*16 (i.e. save area for current window )
           We must ensure that code below works even when a0==a1
        */
        rsr     a2,EXCSAVE_0
        s32i    a2,a1,0         /* save a0 */
        l32i    a2,a0,4
        s32i    a2,a1,4         /* save a1; NOOP if a0==a1 */
        l32i    a2,a0,8
        s32i    a2,a1,8         /* save a2; NOOP if a0==a1 */
        s32i    a3,a1,12        /* save a3 */
        /* Now save other windows.
                We use jump tables to do this.
```

```
        First, we save windows wb+1...n-1 where n == number of windows.
        Second, we save windows 0,...,wb-1
    */
    /* Disable WOE */
    l32r    a3, .LDWOE
    rsr     a2, PS
    and     a2, a2, a3
    wsr     a2, PS
    rsync

    addi    a0,a1,16        /* compute next save area */
    rsr     a1,WINDOWBASE
    slli    a1,a1,2
    l32r    a2,.Laddress_of_save1_table_ptr
    add     a2,a1,a2
    l32i    a2,a2,0
    jx      a2
    /* The instruction jumps into the 1st part of the save sequence
        with the following notable register contents:
        a0 = ar_save_area_ptr + (ipwb+1)*16;  i.e the save area for the
                                              next window.
        wb = ipwb
    */
save1_28: /* ipwb = 0 */
    s32i    a4,a0,0
    s32i    a5,a0,4
    s32i    a6,a0,8
    s32i    a7,a0,12
    addi    a4,a0,16
    rotw    1
save1_24: /* ipwb = 1 */
    s32i    a4,a0,0
    s32i    a5,a0,4
    s32i    a6,a0,8
    s32i    a7,a0,12
    addi    a4,a0,16
    rotw    1
save1_20: /* ipwb = 2 */
    s32i    a4,a0,0
    s32i    a5,a0,4
    s32i    a6,a0,8
    s32i    a7,a0,12
    addi    a4,a0,16
    rotw    1
save1_16: /* ipwb = 3 */
    s32i    a4,a0,0
    s32i    a5,a0,4
    s32i    a6,a0,8
    s32i    a7,a0,12
    addi    a4,a0,16
    rotw    1
save1_12: /* ipwb = 4 */
    s32i    a4,a0,0
    s32i    a5,a0,4
    s32i    a6,a0,8
    s32i    a7,a0,12
    addi    a4,a0,16
    rotw    1
save1_8: /* ipwb = 5 */
    s32i    a4,a0,0
    s32i    a5,a0,4
    s32i    a6,a0,8
    s32i    a7,a0,12
    addi    a4,a0,16
    rotw    1
save1_4: /* ipwb = 6 */
    s32i    a4,a0,0
    s32i    a5,a0,4
    s32i    a6,a0,8
    s32i    a7,a0,12
    rotw    1
```

```
save1_0: /* ipwb = 15 */
        /* at this point, wb = 15; a0 = ar_save_area_ptr+n_aregs*4;
               i.e. a0 points to the end of the save area */
        /* Now save 0...wb-1. i.e. the wrap around case */
        132r    a0,ar_save_ptr
        132r    a2,sr_save_area_ptr
        132i    a1,a2,(WINDOWBASE*4)          /* retrieve ipwb */
        slli    a1,a1,2
        132r    a2,.Laddress_of_save2_table_ptr
        add     a2,a1,a2
        132i    a2,a2,0
        jx      a2
        /* wb = 15; a0 = ar_save_area_ptr */
save2_28: /* ipwb = 7 */
        s32i    a4,a0,0
        s32i    a5,a0,4
        s32i    a6,a0,8
        s32i    a7,a0,12
        addi    a4,a0,16
        rotw    1
save2_24: /* ipwb = 6 */
        s32i    a4,a0,0
        s32i    a5,a0,4
        s32i    a6,a0,8
        s32i    a7,a0,12
        addi    a4,a0,16
        rotw    1
save2_20: /* ipwb = 5 */
        s32i    a4,a0,0
        s32i    a5,a0,4
        s32i    a6,a0,8
        s32i    a7,a0,12
        addi    a4,a0,16
        rotw    1
save2_16: /* ipwb = 4 */
        s32i    a4,a0,0
        s32i    a5,a0,4
        s32i    a6,a0,8
        s32i    a7,a0,12
        addi    a4,a0,16
        rotw    1
save2_12: /* ipwb = 3 */
        s32i    a4,a0,0
        s32i    a5,a0,4
        s32i    a6,a0,8
        s32i    a7,a0,12
        addi    a4,a0,16
        rotw    1
save2_8: /* ipwb = 2 */
        s32i    a4,a0,0
        s32i    a5,a0,4
        s32i    a6,a0,8
        s32i    a7,a0,12
        addi    a4,a0,16
        rotw    1
save2_4: /* ipwb = 1 */
        s32i    a4,a0,0
        s32i    a5,a0,4
        s32i    a6,a0,8
        s32i    a7,a0,12
        rotw    1
save2_0: /* ipwb = 0 */
        /* wb = (ipwb-1) mod (n_aregs/4) */
        /* Now save special registers.
           We key it by testing the presence of register numbers.
           When present, the numbers indicate the user has configured
           the process to have the corresponding processor options.
           Note this doesn't quite work for TIE instructions yets.
        */
        132r    a0,sr_save_area_ptr
```

```
#define SAVE(r)          \
        rsr     a2, r;      \
        s32i    a2, a0, (r*4)

#ifdef ACCLO_OFFSET
        SAVE(ACCLO)
        SAVE(ACCHI)
        SAVE(MR_0)
        SAVE(MR_1)
        SAVE(MR_2)
        SAVE(MR_3)
#endif
#ifdef AV0_OFFSET
        SAVE(AVLO)
        SAVE(AVHI)
        SAVE(BV)
        SAVE(SAV)
#endif
#ifdef BR_OFFSET
        SAVE(BR)
#endif
        SAVE(CACHEATTR)
#ifdef CCOUNT_OFFSET
        SAVE(CCOUNT)
#endif
#ifdef CPENABLE_OFFSET
        SAVE(CPENABLE)
#endif
        SAVE(DEBUGCAUSE)
        SAVE(EPC_1)
        SAVE(EXCSAVE_1)
        SAVE(EXCCAUSE)
        SAVE(ICOUNT)
        SAVE(ICOUNTLEVEL)
        SAVE(INTENABLE)
        SAVE(INTREAD)
        SAVE(LBEG)
        SAVE(LCOUNT)
        SAVE(LEND)
        SAVE(SAR)

        /* Disable Interrupts and Icounts */
        movi.n  a2, 0
        wsr     a2, INTENABLE
        wsr     a2, ICOUNTLEVEL
        wsr     a2, ICOUNT
        isync

         /* Load new PS: Enable  WOE and lower priority.
            We have already turned off interrupts and icount
            from above. */
         l32r    a1 ,.Lps
        wsr     a1, PS

        movi.n  a0, 0
        movi.n  a2, 1
        wsr     a2, WINDOWSTART              /* window start = 1 */
        wsr     a0, WINDOWBASE         /* window base = 0 */
        rsync

        /* Initialize our stack and call handler */
        l32r    a1, _xmon_stack_ptr
        l32r    a2, handler
        callx4  a2

        /* Raise interrupt level back up and disable WOE */
        rsr     a2, PS
        movi.n  a3, 0
        or      a2, a2, a3
         l32r    a3, .LDWOE
         and     a2, a2, a3
```

```
        wsr     a2, PS
        rsync

        /* restore sequence */
        l32r    a0, sr_save_area_ptr

#define RESTORE(r)          .\
        l32i    a2, a0, (r*4);      \
        wsr     a2, r

#ifdef ACCLO_OFFSET
        RESTORE(ACCLO)
        RESTORE(ACCHI)
        RESTORE(MR_0)
        RESTORE(MR_1)
        RESTORE(MR_2)
        RESTORE(MR_3)
#endif
#ifdef AV0_OFFSET
        RESTORE(AVLO)
        RESTORE(AVHI)
        RESTORE(BV)
        RESTORE(SAV)
#endif
#ifdef BR_OFFSET
        RESTORE(BR)
#endif
        RESTORE(CACHEATTR)
#ifdef CCOUNT_OFFSET
        RESTORE(CCOUNT)
#endif
#ifdef CPENABLE_OFFSET
        RESTORE(CPENABLE)
#endif
        RESTORE(EPC_1)
        RESTORE(EXCSAVE_1)
        RESTORE(EXCCAUSE)
        RESTORE(INTENABLE)
        RESTORE(INTREAD)
        RESTORE(LBEG)
        RESTORE(LCOUNT)
        RESTORE(LEND)
        RESTORE(SAR)


        /* Now restore all the ar's */
        l32i    a2,a0,(WINDOWBASE*4)
        wsr     a2,WINDOWBASE           /* set wb to ipwb */
        rsync
        l32r    a0,ar_save_ptr
        rsr     a2,WINDOWBASE
        slli    a2,a2,2                 /* multiply by 4 */
        l32r    a3,.Laddress_of_restore1_table_ptr
        add     a3,a2,a3
        l32i    a3,a3,0
        slli    a2,a2,2                 /* multiply by 4 */
        add     a8,a0,a2
        addi    a8,a8,16
        jx      a3
        /* wb = ipwb; a8 = ar_save_area_ptr + (ipwb+1)*16 */
restore1_28: /* ipwb = 0 */
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        addi    a12,a8,16
        rotw    1
restore1_24: /* ipwb = 1 */
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
```

```
        l32i    a7,a8,12
        addi    a12,a8,16
        rotw    1
restore1_20: /* ipwb = 2 */
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        addi    a12,a8,16
        rotw    1
restore1_16: /* ipwb = 3 */
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        addi    a12,a8,16
        rotw    1
restore1_12: /* ipwb = 4 */
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        addi    a12,a8,16
        rotw    1
restore1_8: /* ipwb = 5 */
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        addi    a12,a8,16
        rotw    1
restore1_4: /* ipwb = 6 */
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        rotw    1
restore1_0: /* ipwb = 15 */
        /* wb = 15 */
        /* restore window 0...wb-1 or none if wb == 0 */
        l32r    a4, sr_save_area_ptr
        l32i    a5,a4,(WINDOWBASE*4)
        slli    a5,a5,2
        l32r    a6,.Laddress_of_restore2_table_ptr
        add     a6,a5,a6
        l32i    a6,a6,0
        l32r    a4,ar_save_ptr
        addi    a4,a4,-16
        jx      a6
        /* wb = 15; a4 = ar_save_area_ptr-16 */
restore2_28: /* ipwb = 7 */
        addi    a8,a4,16
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        rotw    1
restore2_24: /* ipwb = 6 */
        addi    a8,a4,16
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        rotw    1
restore2_20: /* ipwb = 5 */
        addi    a8,a4,16
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        rotw    1
```

```
restore2_16: /* ipwb = 4 */
        addi    a8,a4,16
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        rotw    1
restore2_12: /* ipwb = 3 */
        addi    a8,a4,16
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        rotw    1
restore2_8: /* ipwb = 2 */
        addi    a8,a4,16
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        rotw    1
restore2_4: /* ipwb = 1 */
        addi    a8,a4,16
        l32i    a4,a8,0
        l32i    a5,a8,4
        l32i    a6,a8,8
        l32i    a7,a8,12
        rotw    1
restore2_0:
        l32i    a5,a4,20
        l32i    a6,a4,24
        l32i    a7,a4,28
        l32i    a4,a4,16
        rotw    1

        /* set wstart to what the user had */
        wsr     a0,EXCSAVE_0
        l32r    a0, sr_save_area_ptr

        l32i    a0, a0, (WINDOWSTART*4)
        wsr     a0,WINDOWSTART
        rsr     a0, WINDOWBASE
        wsr     a0, WINDOWBASE /* no-op but to avoid iss problem */

        l32r    a0, ar_save_ptr
        s32i     a1, a0, 0  /* save a1, we don't need loc anymore*/

        /* restore ICOUNT & ICOUNTLVL */
        l32r    a0, sr_save_area_ptr
        movi.n  a1, 0
        wsr     a1, ICOUNTLEVEL        // first lower icountlevel to 0
        isync
        l32i    a1, a0, (ICOUNT*4)
        wsr     a1, ICOUNT      // now write icount.
        isync
        l32i    a1, a0, (ICOUNTLEVEL*4)
        wsr     a1, ICOUNTLEVEL        // finally set icountlvl
        isync

        /* Enable  WOE */
        //l32r    a1 ,.LEWOE
        //rsr   a0, PS
        //or    a0, a0, a1
        //wsr   a0, PS
        //rsync
        //l32r a0, save_area_ptr

        // Put ar_save_area_ptr back into a0 so
        // that we can restore a1
        l32r    a0, ar_save_ptr
        l32i    a1, a0,0
```

```
        rsr     a0,EXCSAVE_0

        rfi     0

        .align  4
_flush_i_cache:
        entry   sp, 48
        dhwb    a2, 0   /* force it out of the data cache (if present) */
        /* Use ihi for a little more efficiency */
        ihi     a2, 0   /* invalidate in i-cache (if present) */
        isync           /* just for safety sake */
        retw.n

// Functions to help us out when running inside simulator.

        .align  4
_xmon_out:
        entry   sp,16
        mov.n   a3,a2   // pass the 2nd arg as the first arg.
        movi.n  a2,-2   // sys_xmon_out
        or      a4,a4,a4 // force window overflow before simcall
        simcall
        retw.n

        .align  4
_xmon_in:
        entry   sp,16
        movi.n  a2,-3
        or      a4,a4,a4 // force window overflow before simcall
        simcall
        retw.n

        .align  4
_xmon_flush:
        entry   sp,16
        movi.n  a2,-4
        or      a4,a4,a4 // force window overflow before simcall
        simcall
        retw.n
        .align 4
_xmon_init:
        entry   sp,16
        movi.n  a2, -7
        or      a4,a4,a4 // force window overflow before simcall
        simcall
        retw.n
        .align 4
.global _xmon_crash
_xmon_crash:
        entry   sp, 16
        .byte   0,0,0,0,0,0
        retw.n




# unsigned _xmon_get_cpenable()
#

        .global _xmon_get_cpenable
        .align  4
_xmon_get_cpenable:
        entry   sp, 16
#ifdef CPENABLE_OFFSET
        rsr     a2, CPENABLE
#endif
        retw
```

```
# void _xmon_set_cpenable(unsigned value)
#
# a2 -- holds the value to set cpenable to
#

        .global _xmon_set_cpenable
        .align 4
_xmon_set_cpenable:
        entry   sp, 16
#ifdef CPENABLE_OFFSET
        wsr     a2, CPENABLE
        rsync
#endif
        retw



# void _xmon_set_user_register(unsigned user_register, unsigned value, unsigned *execute_here)
#       a2 -- user_register
#       a3 -- value
#       a4 -- pointer to memory to execute from

#
        .align 4


.wur0_instruction:
        .word   0x00f30000
.wur0_insn_ptr:
        .word   .wur0_instruction

.wur0_placeholder_ptr:
        .word   .wur0_instruction_placeholder


        .align 4
.global _xmon_set_user_register
_xmon_set_user_register:
        entry   sp, 48


# a6 -- temporary for moving memory
# a5 -- pointer to wur0_placeholder
# a4 -- points to the RAM location we will
#       execute from, move the base instruction
#       (including the retw) to that point.

        l32r    a5, .wur0_placeholder_ptr
        l32i    a6, a5, 0
        s32i    a6, a4, 0
        l32i    a6, a5, 4
        s32i    a6, a4, 4

# a5 -- available again, now used to load the
#       base wur instruction which we will now
#       modify for the correct ar and user register
#       number
# a6 -- holds the modified instruction


        l32r    a5, .wur0_insn_ptr
        l32i    a6, a5, 0

# a2 -- holds the user register we are going to write
# a4 -- holds the location in memory that we are going
#       to execute from
# a6 -- holds the instruction we are going to execute
```

```
        slli    a2, a2, 8
        or      a6, a6, a2

# a2 -- Can be used as a temporary now, to
#       OR in the 3, which is the register that
#       holds the value we are going to write to
#       the user register

        movi    a2, 3
        slli    a2, a2, 4
        or      a6, a6, a2

# a2 -- Temporary for merging instructions
# a4 -- pointer to the location we are going to execute
#       from
# a5 -- Holds the value we load from our execution point
# a6 -- The instruction that we are going to execute

        l32i    a5, a4, 0

# Need to merge our 24-bit instruction with 8 bits
# from our execute point
# Want to use the lower 24 bits from a6,
# and the upper 8-bits from a5

        movi    a2, 0xff
        slli    a2, a2, 24
        and     a5, a5, a2
        or      a6, a6, a5
        s32i    a6, a4, 0

# Flush the cache
        mov     a10, a4
        call8   _flush_i_cache

        jx      a4


# Want the upper 24-bits from a6, and the
# lower 8-bits from a4



.align 4
.wur0_instruction_placeholder:
        or      a0, a0, a0
        retw



#
#
# Data for _xmon_get_user_register
#
#

        .align 4
.rur0_insn:
        .word   0x00e30000

.rur0_insn_ptr:
        .word   .rur0_insn


.rur_placeholder_ptr:
        .word   .rur_instruction_placeholder


# unsigned int  _xmon_get_user_register(unsigned user_register, unsigned *execute_here)
#
```

```
#       a2(input)  -- user_register
#       a2(output) -- contains the value
#       a3(input)  -- address for executing instructions

        .align  4
.global _xmon_get_user_register
_xmon_get_user_register:
        entry   sp, 48

# a5 -- temporary for moving memory
# a4 -- Points to our rur instruction including ret
#       that we are going to copy to the execution point
# a3 -- Points to the execution point

        l32r    a4, .rur_placeholder_ptr
        l32i    a5, a4, 0
        s32i    a5, a3, 0
        l32i    a5, a4, 4
        s32i    a5, a3, 4


# a4 -- Temp that Points to the rur0 instruction
# a6 -- will hold the rur instruction throughout

        l32r    a4, .rur0_insn_ptr
        l32i    a6, a4, 0

# Shift the user register number to the correct
# offset and OR it into our instruction
# a2 -- Holds the user register being read
# a6 -- instruction being massgaed

        slli    a2, a2, 4
        or      a6, a6, a2

# Now need to set the r-field of the instruction
# to be 2, which is the return value of this function
# a5 -- Temp that holds the constant being ord in
# a6 -- The instruction being massaged

        movi    a5, 2
        slli    a5, a5, 12
        or      a6, a6, a5

# Now load in the word from where we are going to execute
# the rur, merge our rur instruction, and store that word
# back to memory.
# a2 -- Temp for masking
# a3 -- Points to the correct memory location
# a5 -- Holds the WORD we are manipulating

        l32i    a5, a3, 0

# In Little Endian we save the MSB and put our
# instruction in the lower 3 bytes

        movi    a2, 0xff
        slli    a2, a2, 24
        and     a5, a5, a2
        or      a6, a6, a5
        s32i    a6, a3, 0

# Clear the cache line
# a3 -- Points to the location being cleared

        mov     a6, a3
        call4   _flush_i_cache

        movi    a2, 0
        jx      a3
```

```
# A place holder that will be dynamically replaced with
# the correct rur instruction

        .align  4
.rur_instruction_placeholder:
        or      a0, a0, a0
        retw.n



        .global g_dummy_entry_instruction
        .global g_dummy_retw_instruction
        .global g_dummy_entry_ptr
        .global g_dummy_retw_ptr

        .align  4
g_dummy_entry_instruction:
        entry   sp, 16

        .align  4
g_dummy_retw_instruction:
        retw.n



        .align  4
g_dummy_entry_ptr:
        .word   g_dummy_entry_instruction

        .align  4
g_dummy_retw_ptr:
        .word   g_dummy_retw_instruction


# void _xmon_execute_here(unsigned a4_value, void *execute_here);
#
# a2 -- value to be stuffed into a4
# a3 -- execute the instructions at this address
#

        .global _xmon_execute_here
        .align  4
_xmon_execute_here:
        entry   sp, 16

# a8 will be the a4 value after the call4 to the address
        mov     a8, a2
        callx4  a3
        retw
```

II. Xtensa-mon.c

```
/*****************************************************************************
 *************
 *
 *    The following gdb commands are supported:
 *
 * command          function                                Return value
 *
 *    g             return the value of the CPU registers   hex data or ENN
 *    G             set the value of the CPU registers      OK or ENN
 *
 *    mAA..AA,LLLL  Read LLLL bytes at address AA..AA       hex data or ENN
 *    MAA..AA,LLLL: Write LLLL bytes at address AA.AA       OK or ENN
```

```
 *
 *    c             Resume at current address              SNN   ( signal NN)
 *    cAA..AA       Continue at address AA..AA             SNN
 *
 *    s             Step one instruction                   SNN
 *    sAA..AA       Step one instruction from AA..AA       SNN
 *
 *    k             kill
 *
 *    ?             What was the last sigval ?             SNN   (signal NN)
 *
 *    bBB..BB      Set baud rate to BB..BB              OK or BNN, then sets
 *                                                      baud rate
 *
 * All commands and responses are sent with a packet which includes a
 * checksum.  A packet consists of
 *
 * $<packet info>#<checksum>.
 *
 * where
 * <packet info> :: <characters representing the command or response>
 * <checksum>    :: < two hex digits computed as modulo 256 sum of <packetinfo>>
 *
 * When a packet is received, it is first acknowledged with either '+' or '-'.
 * '+' indicates a successful transfer.  '-' indicates a failed transfer.
 *
 * Example:
 *
 * Host:                  Reply:
 * $m0,10#2a               +$000102030405060708091011121314l5#42
 *
 ****************************************************************************/

#include <stdio.h>
#include <signal.h>
#include <machine/specreg.h>
#include <machine/xt1000.h>
#include "DebugExceptionVectorHandler.h"
#include "uart.h"
#include "xtensa-libdb.h"

#define WS_MASK (~((~0)<<(NUM_AREGS/4)))
#ifdef IS_LITTLE_ENDIAN

#define IS_BREAKN(p) ((p)[0]==0x2d && ((p)[1]&0xf0)==0xf0)
#define IS_BREAK(p)  ((p)[2]==0x00 && ((p)[0]&0x0f)==0x00 && ((p)[1]&0xf0)==0x40)
#define BREAKNO(p)   (IS_BREAKN(p) ? ((p)[1]&0x0f) : -1 )
#define BREAK_S(p)   ((p)[1]&0x0f)
#define BREAK_T(p)   (((p)[0]&0xf0)>>4)
#else

#define IS_BREAKN(p) ((p)[0]==0xd2 && ((p)[1]&0x0f)==0x0f)
#define IS_BREAK(p)  ((p)[2]==0x00 && ((p)[Q]&0xf0)==0x00 && ((p)[1]&0xf)==0x04)
#define BREAKNO(p)   (IS_BREAKN(p) ? (((p)[1]&0xf0)>>4) : -1 )
#define BREAK_S(p)   (((p)[1]&0xf0)>>4)
#define BREAK_T(p)   ((p)[0]&0x0f)

#endif

#define SR_REG(n) (_sr_registers[ (n) ])

/* Macros to extract fields of PS */
#define GET_PSINTLVL(ps)  ((ps)&0xf)
#define GET_PSUSRMODE(ps) (((ps)>>5)&0x1)
#define GET_PSOWB(ps)     (((ps)>>8)&0xf)
#define GET_PSCALLINC(ps) (((ps)>>16)&0x3)
#define GET_PSWOE(ps)     (((ps)>>18)&0x1)

/* Imported functions */
extern void _flush_i_cache( char *);
extern int _xmon_out(char c);
```

```
extern int _xmon_in( void );
extern int _xmon_flush( void );
extern void xmon_init( void );

/* forward definitions */
static long reg_at_wb( unsigned int reg, unsigned int wb );
static int reg_at_ipwb( unsigned int reg );
static int save_to_stack(); /* returns 0 on success, -1 on error */
static void mon_error( char *);
static void putDebugChar(char);   /* write a single character      */
static int getDebugChar();   /* read and return a single char */
static void putDebugString(char *);
static int flushDebug();
```

```
/* Parameters:
   We'll need to create a configuration process that generates
   many of these defines.
*/
/*************************************************************************/
/* BUFMAX defines the maximum number of characters in inbound/outbound buffers*/
/* at least NUMREGBYTES*2 are needed for register packets */
#define BUFMAX 2048

#define AR_SAVE_SIZE (4*NUM_AREGS)
#define SR_SAVE_SIZE (4*256)

#define PC EPC_0
#define DBG_EPS EPS_0

long _ar_registers[AR_SAVE_SIZE/(sizeof (long))];
long _sr_registers[SR_SAVE_SIZE/(sizeof (long))];


/* !0 means we are running on the board, defined by linker */
extern void *IN_SIMULATOR;
int _in_simulator = (int)&IN_SIMULATOR;
int _initialized = 0;  /* !0 means we've been initialized */


static const char hexchars[]="0123456789abcdef";

/* string functions */
int _strlen( char *cp )
{
  int i;
  if( cp == 0 )
    return 0;
  i = 0;
  while ( *cp )
    {
      i++;
      cp++;
    }
  return i;
}

char *_strcpy( char *d, char *s )
{
  char *cp = d;
  if( d && s )
    {
      while( *s )
        {
          *cp = *s;
          cp++;
          s++;
        }
      *cp = 0;
    }
  return d;
}


void _memset(unsigned char *ptr, unsigned char value, int num)
{
  while (num > 0)
  {
    *ptr = value;
    ++ptr;
    --num;
  }
}
```

```
/* Log errors for transmission */
#define LOG_SIZE 100
static char *error_log_end;
static char error_log[LOG_SIZE];

static void mon_error_clear()
{
  error_log_end = error_log;
  error_log_end[0] = 0;
}

static void mon_error( char *msg )
{
  if( error_log_end == 0 )
    error_log_end = error_log;
  while( *msg )
    {
      if( error_log_end < &error_log[LOG_SIZE-1] )
        *error_log_end++ = *msg++;
      else
        break;
    }
  error_log_end[0] = 0;
}

/* Convert ch from a hex digit to an int */

static int
hex(ch)
     unsigned char ch;
{
  if (ch >= 'a' && ch <= 'f')
    return ch-'a'+10;
  if (ch >= '0' && ch <= '9')
    return ch-'0';
  if (ch >= 'A' && ch <= 'F')
    return ch-'A'+10;
  return -1;
}

/* scan for the sequence $<data>#<checksum>     */

static void
getpacket(buffer)
     char *buffer;
{
  unsigned char checksum;
  unsigned char xmitcsum;
  int i;
  int count;
  unsigned char ch;

  do
    {
      /* wait around for the start character, ignore all other characters */
      while ((ch = (getDebugChar() & 0x7f)) != '$') ;

      checksum = 0;
      xmitcsum = -1;

      count = 0;

      /* now, read until a # or end of buffer is found */
      while (count < BUFMAX)
        {
          ch = getDebugChar() & 0x7f;
          if (ch == '#')
```

```
            break;
          checksum = checksum + ch;
          buffer[count] = ch;
          count = count + 1;
        }

      if (count >= BUFMAX)
       continue;

      buffer[count] = 0;

      if (ch == '#')
       {
         xmitcsum = hex(getDebugChar() & 0x7f) << 4;
         xmitcsum |= hex(getDebugChar() & 0x7f);
#if 0
         /* Humans shouldn't have to figure out checksums to type to it. */
         putDebugChar ('+');
         return;
#endif
         if (checksum != xmitcsum)
           putDebugChar('-'); /* failed checksum */
         else
           {
             putDebugChar('+'); /* successful transfer */
             /* if a sequence char is present, reply the sequence ID */
             if (buffer[2] == ':')
               {
                 putDebugChar(buffer[0]);
                 putDebugChar(buffer[1]);
                 /* remove sequence chars from buffer */
                 count = _strlen(buffer);
                 for (i=3; i <= count; i++)
                   buffer[i-3] = buffer[i];
               }
           }
         flushDebug();
       }
    }
  while (checksum != xmitcsum);
}

/* send the packet in buffer.  */

/* Convert the memory pointed to by mem into hex, placing result in buf.
 * Return a pointer to the last char put in buf (null), in case of mem fault,
 * return 0.
 * If MAY_FAULT is non-zero, then we will handle memory faults by returning
 * a 0, else treat a fault like any other fault in the stub.
 */

static unsigned char *
mem2hex(mem, buf, count, may_fault)
     unsigned char *mem;
     unsigned char *buf;
     int count;
     int may_fault;
{
  unsigned char ch;
  while (count-- > 0)
    {
      ch = *mem++;
      *buf++ = hexchars[ch >> 4];
      *buf++ = hexchars[ch & 0xf];
    }

  *buf = 0;


  return buf;
}
```

```
static void
putpacket(buffer)
     unsigned char *buffer;
{
  unsigned char checksum;
  unsigned char ack;
  int count;
  unsigned char ch;

  /*  $<packet info>#<checksum>. */
  do
    {
      putDebugChar('$');
      checksum = 0;
      count = 0;

      while (ch = buffer[count])
        {
          putDebugChar(ch);
          checksum += ch;
          count += 1;
        }

      putDebugChar('#');
      putDebugChar(hexchars[checksum >> 4]);
      putDebugChar(hexchars[checksum & 0xf]);

      flushDebug();
      ack = getDebugChar();
      ack = ack & 0x7f;

//      led_display_ok();
      /*
      if (ack != '+')
      {
        //        char buf[8];
        //      _memset(buf, 0, sizeof(buf));
        //      putDebugString("--");
        //      mem2hex(&ack, buf, 1, 0);
         putDebugString(buf);
        putDebugString("--");
      }
      else
        putDebugChar('Y');
      */
    }
  while (ack != '+');
}

static char remcomInBuffer[BUFMAX];
static char remcomOutBuffer[BUFMAX];

static unsigned char g_execute_here[1024];


static void bad_protocol()
{
  _strcpy( remcomOutBuffer, "Error: garbled command" );
}

static void aok()
{
  _strcpy( remcomOutBuffer, "OK" );
}

/* Decode a hex string and write it into memory. */
static char *
write_mem(buf, mem, count, flush, verify)
     register unsigned char *buf;
     register unsigned char *mem;
```

```
    int count;
    int flush;
    int verify;
{
  int i;
  unsigned char ch;
  unsigned char *start = mem;

  for (i=0; i<count; i++)
    {
      ch = hex(*buf++) << 4;
      ch |= hex(*buf++);
      *mem = ch;
      if( verify && *mem != ch )
        return 0;
      mem += 1;
    }

  if( flush ) {
    while( count >= 0 ) {
      _flush_i_cache( start );
      count -= 4;
      start += 4;
    }
    /* we do one more flush just in case the last
       instruction straddled two cached line */
    _flush_i_cache( start );
  }

  return mem;
}


/*
 * While we find nice hex chars, build an int.
 * Return number of chars processed.
 */

static int
hexToInt(char **ptr, int *intValue)
{
  int numChars = 0;
  int hexValue;

  *intValue = 0;

  while (**ptr)
    {
      hexValue = hex(**ptr);
      if (hexValue < 0)
        break;

      *intValue = (*intValue << 4) | hexValue;
      numChars ++;

      (*ptr)++;
    }

  return (numChars);
}


static void set_icount_for_single_step(int intlevel)
{
    /* set the icount level to one more than the interrupt level,
       This will allow single-stepping through handlers */
    SR_REG(ICOUNT) = -2;
    SR_REG(ICOUNTLEVEL) = intlevel < DEBUG_INTERRUPT_LEVEL ? intlevel+1 :
        DEBUG_INTERRUPT_LEVEL;
}
```

```
/* The _handle_exception function is best modeled as a state machine */
static int state;               /* zero'ed by during bss initialization */
                                  /* Thus XMON_INITIAL must be zero. */
#define XMON_INITIAL 0          /* start up xmon */
#define XMON_CONTROL 1          /* xmon is stopped, polling
                                     commands from host */
#define XMON_RUNNING 2            /* xmon is running, waiting for
                                     an external event */
#define XMON_RESUMING  3        /* an interrupt, not the serial
                                     interrupt, has occurred. */

/* Data structe to keep track of hw breakpoints */
struct hw_break_info {
  int free;
  char *addr;
  int reg_number;
} hw_break[NIBREAK] = HW_BREAK_INIT;

/* special breaks are how we detect SIGINT and SIGILL */
typedef void (*special_breakpoint_handler)(int *,int *);
static void sigint_handler(int *,int*);
static void _tell_gdb(int);

struct special_breakpoint {
  char *address;
  special_breakpoint_handler f;
  char saved_inst[3];
} special_break[] = {
  { (char *)UART_VECTOR, sigint_handler },
#ifdef UART_SECOND_VECTOR
  { (char *)UART_VECTOR_2, sigint_handler },
#endif
  { (char *)0, (special_breakpoint_handler)0}
};

static void init_special_breaks()
{
  /* nothing to do right now */
}

static void set_special_breaks()
{
  struct special_breakpoint *b;
  b = special_break;
  while( b->address )
    {
#ifdef ISAUSEDENSITYINSTRUCTION
      b->saved_inst[0] = b->address[0];
      b->saved_inst[1] = b->address[1];
#ifdef IS_LITTLE_ENDIAN
      b->address[0] = 0x2d;
      b->address[1] = 0xf1;
#else
      b->address[0] = 0xd2;
      b->address[1] = 0x1f;
#endif
      _flush_i_cache(b->address);
      _flush_i_cache(b->address+1);
#else
      b->saved_inst[0] = b->address[0];
      b->saved_inst[1] = b->address[1];
      b->saved_inst[2] = b->address[2];
#ifdef IS_LITTLE_ENDIAN
      b->address[0] = 0x10;
      b->address[1] = 0x40;
      b->address[2] = 0x00;
#else
      b->address[0] = 0x01;
      b->address[1] = 0x04;
      b->address[2] = 0x00;
```

```
#endif
        _flush_i_cache(b->address);
        _flush_i_cache(b->address+1);
        _flush_i_cache(b->address+2);
#endif
        b++;
      }
}

static void clear_special_breaks()
{
  struct special_breakpoint *b;
  b = special_break;
  while( b->address )
    {
#ifdef ISAUSEDENSITYINSTRUCTION
        b->address[0] = b->saved_inst[0];
        b->address[1] = b->saved_inst[1];
        _flush_i_cache(b->address);
        _flush_i_cache(b->address+1);
#else
        b->address[0] = b->saved_inst[0];
        b->address[1] = b->saved_inst[1];
        b->address[2] = b->saved_inst[2];
        _flush_i_cache(b->address);
        _flush_i_cache(b->address+1);
        _flush_i_cache(b->address+2);
#endif
        b++;
      }
}

// !!@


static void do_special_breaks(int *state, int *sigval)
{
  char *pc;
  struct special_breakpoint *b;

  b = special_break;
  pc = (char *)SR_REG(PC);


  while( b->address )
  {
      if( b->address == pc && b->f )
      {
          b->f(state, sigval);
          return;
      }
      b++;
  }

  *state = XMON_CONTROL;
  *sigval = SIGTRAP;
}


#if 0
void set_ps( int eps )
{
  REG(PSINTLVL)   = GET_PSINTLVL(eps);
  REG(PSUSRMODE)  = GET_PSUSRMODE(eps);
  REG(PSOWB)      = GET_PSOWB(eps);
  REG(PSCALLINC)  = GET_PSCALLINC(eps);
  REG(PSWOE)      = GET_PSWOE(eps);
}
#endif
```

```
#if 0
void flash_value(unsigned int value)
{
    int i = 0;

    for(i = 28; i >= 0; i=i-4)
    {
      int number = (value >> i) & 0xf;

      led_blank();
      led_pause(100000);
      led_display_digit(number);
      led_pause(100000);
    }
}
#endif


void setup_ps()
{
/* the code to set up the PS works quite differently depending
   on whether or not the uart is on interrupt level one. */
#if UART_INTERRUPT_LEVEL == 1
      {
       unsigned int real_ps = 0;
       real_ps = SR_REG(EPS_0);

       // We can figure out if PS.UM was set by looking
       // at which vector we came from
       // UART_VECTOR is actually the UserExceptionVector

       if ( SR_REG(UART_EPC) == UART_VECTOR )
       {
         // Since we are coming from UserExceptionVector
         // turn on the PS.UM mode bit.
         real_ps = real_ps | (1 << 4);

         // Assume that WOE is always enabled for user code.
         real_ps = real_ps | (1 << 17);
       }
       else
       {
         // We are coming from the KernelExceptionVector
         // So we leave PS.UM disabled, and we take
          // WOE from the current PS.
         real_ps = real_ps | (  SR_REG(EPS_0) & (1 << 17)  );
       }

       // Set interrupt level to 0
       real_ps = real_ps & ~0xf;
       SR_REG(EPS_0) = real_ps;
      }
#elif UART_INTERRUPT_LEVEL != -1
     SR_REG(EPS_0) = SR_REG(UART_EPS);
#endif
}


/* If we see a break on the UART then we simulatre a SIGINT */
static void sigint_handler( int *state, int *sigval )
{
  int interrupts = SR_REG(INTERRUPT);
  int c;

//  led_display_digit(7);
//  led_pause(100000);

//  flash_value(interrupts);
```

```
  if( interrupts & UART_INTERRUPT )
  {
//      led_display_digit(8);
//      led_pause(100000);

      /* the received interrupt was the serial interrupt for the
         port that is being used for xmon control. GDB has
         requested that xmon break. */
      *state = XMON_CONTROL;
      c = getDebugChar();     /* read the break char, to avoid
                                 gdb protocol sync problems */
      *sigval = SIGINT;               /* indicate SIGINT */

      /* unwind the interrupt: set up registers so that it appears
         we returned from interrupt handler.
      */

      // !!@
//      led_display_digit(9);
//      led_pause(100000);

#if UART_INTERRUPT_LEVEL != -1
      SR_REG(PC) = SR_REG(UART_EPC);
#endif
      setup_ps();
  }
  else
  {
      /* this is some other level2 interrupt. Special breaks were
         already cleared so set the state into resume mode and
         set the icount up for a single instruction. This will
         allow us to step over the instruction and restore the
         break.*/
//      led_display_digit(10);
//      led_pause(100000);

      *state = XMON_RESUMING;
      set_icount_for_single_step( SR_REG(DBG_EPS) & 0x0f );
  }
}



void
_handle_exception()
{
    int n;
    int sigval = SIGTRAP;
    unsigned char *pc;
    pc = (unsigned char *)SR_REG(PC);

    /* reset icount, so that we can continue, in case
       we came here because of an icount interrupt */
    SR_REG(ICOUNT) = 0;
    SR_REG(ICOUNTLEVEL) = 0;

    /* when we return enable all interrupts except timers */
#ifdef TIMER_INTERRUPT_MASK
//    SR_REG(INTENABLE) = ALL_INTERRUPT_MASK & (~TIMER_INTERRUPT_MASK);
    SR_REG(INTENABLE) = ALL_INTERRUPT_MASK;
#else
    SR_REG(INTENABLE) = ALL_INTERRUPT_MASK;
#endif

    for(;;)
    {
        switch( state )
        {
        case XMON_INITIAL:
            if( !_in_simulator )
            {
```

```
            /* We're running on the board, so flash the status
               LEDs a few times */
            _uart_init((uart_dev_t *)XT1000_DUART_1_ADDR, B38400);
            _uart_enable_rcvr_int((uart_dev_t *)XT1000_DUART_1_ADDR);
            led_display_ok();
//          led_display_digit(2);
        }
        else
        {
            _xmon_init();
        }
        putDebugString("XMON R2.5 ");
        putDebugString( _in_simulator ? " running on iss\n"
                      : " running on eval board\r\n");

        _initialized = 1;

        init_special_breaks();
        state = XMON_CONTROL;
        continue;

    case XMON_CONTROL:     /* let host control us */
        _tell_gdb( sigval );
        set_special_breaks();
        state = XMON_RUNNING;
        return;            /* return to user program */

    case XMON_RUNNING:
        if(IS_BREAK(pc)) {
            unsigned s = BREAK_S(pc);
            unsigned t = BREAK_T(pc);
            if( s==1 && (t <= 1)) {
                switch(SR_REG(EXCCAUSE)) {
                case EXCCAUSE_ILLEGAL:
                    sigval = SIGILL;
                    break;
                case EXCCAUSE_SYSCALL:
                    sigval = SIGTRAP;
                    break;
                case EXCCAUSE_IFETCHERROR:
                case EXCCAUSE_LOADSTOREERROR:
                    sigval = SIGSEGV;
                    break;
                case EXCCAUSE_LEVEL1INTERRUPT:
                    sigval = SIGINT;
                    break;
                }
                clear_special_breaks();
                state = XMON_CONTROL;

                /* pretend as though we caught it
                   at the point of occurence */
                // !!@
                SR_REG(PC) = SR_REG(EPC_1);
                setup_ps();
                continue;
            }
        }
        n = BREAKNO(pc);
        switch(n)
        {
        default:
            clear_special_breaks();
            state = XMON_CONTROL;
            break;

        case 1:            /* special breakpoint */
            clear_special_breaks();

            /* keep in mind that the state can be changed by the
               do_special_breaks handler. */
```

```
                do_special_breaks(&state, &sigval);
                break;
            }
            continue;

        case XMON_RESUMING:
            /* we have taken an interrupt that was not the serial
               interrupt and have now executed the original instruction
               at the interrupt vector. Now restore the break instruction
               so that interrupts continue to work and resume. */

            set_special_breaks();
            state = XMON_RUNNING;
            return;
        }
    }
}

/*
 * This function does all command procesing for interfacing to gdb.
 */
unsigned char dummy[4];
unsigned int  user_register_value;

unsigned int execution_space[2];


static unsigned char *
get_reg_ptr(const unsigned int libdb_target_number)
{
  unsigned char *reg_ptr = NULL;
  unsigned      old_cpe  = 0;
  int           offset   = 0;

  switch( GET_TARGET_REG_TYPE(libdb_target_number) )
    {
    case REGTYPE_AR:
      offset = GET_TARGET_REG_INDEX(libdb_target_number);
      reg_ptr = (unsigned char *)&_ar_registers[offset];
      break;

    case REGTYPE_SPECIAL_REG:
      offset = GET_TARGET_REG_INDEX(libdb_target_number);
      reg_ptr = (unsigned char *)&_sr_registers[offset];
      break;

    case REGTYPE_USER_REG:
      old_cpe = _xmon_get_cpenable();
      _xmon_set_cpenable((unsigned)-1);
      user_register_value = _xmon_get_user_register( GET_TARGET_REG_INDEX(libdb_target_number),
                                                     &execution_space[0] );
      _xmon_set_cpenable(old_cpe);
      reg_ptr = (unsigned char *)&user_register_value;
      break;

    default:
      reg_ptr = NULL;
      break;
    }

  return reg_ptr;
}


static unsigned int
set_reg_value(const unsigned int libdb_target_number, const unsigned int value)
{
  int success = 0;

  if ( GET_TARGET_REG_TYPE(libdb_target_number) == REGTYPE_USER_REG )
    {
```

```
    unsigned int old_cpe = 0;

    old_cpe = _xmon_get_cpenable();
    _xmon_set_cpenable((unsigned int)-1);

    _xmon_set_user_register( GET_TARGET_REG_INDEX(libdb_target_number),
                             value,
                             &execution_space[0]);

    _xmon_set_cpenable( old_cpe );
    success = 1;
  }
 else
  {
    unsigned char *reg_ptr = get_reg_ptr(libdb_target_number);
    if (reg_ptr != NULL)
     {
       long *tmp_ptr = (long *)reg_ptr;
       *tmp_ptr = value;
       success = 1;
     }
    else
     {
       success = 0;
     }
  }

 return success;
}



extern unsigned char *g_dummy_entry_ptr;
extern unsigned char *g_dummy_retw_ptr;

typedef void (*FPTR)(void);

static int
ExecuteSomeInstruction(char *pInstruction)
{
  // Execute an instruction, A4 has been setup to point
  // at the spill location (a4 is in the ar_registers)

  // The length of the instruction is coded by the number
  // of characters being passed down.

  unsigned int  dummy              = 0;
  unsigned int  converted          = 0;
  unsigned int  index              = 0;
  unsigned int  a4_value           = 0;
  unsigned int  wb                 = 0;
  unsigned int  a4_index           = 0;
  int           success            = 0;
  unsigned int old_cpe                    = 0;

 old_cpe = _xmon_get_cpenable();
 _xmon_set_cpenable((unsigned int)-1);

  if (pInstruction == NULL)
    goto exit_gracefully;


  wb = SR_REG(WINDOWBASE);
  a4_index = ((4 * wb) + 4) % NUM_AREGS;
  a4_value = _ar_registers[a4_index];


  // skip the initial ':'


  g_execute_here[0] = g_dummy_entry_ptr[0];
```

```
  g_execute_here[1] = g_dummy_entry_ptr[1];
  g_execute_here[2] = g_dummy_entry_ptr[2];

  index = 3;

  _flush_i_cache( (char *)&g_execute_here[0] );

  while (pInstruction && *pInstruction)
    {
      // Skip the ':' characters

      ++pInstruction;
      converted = hexToInt(&pInstruction, &dummy);
      if (converted != 2)
        goto exit_gracefully;

      g_execute_here[index] = (unsigned char)dummy;
      _flush_i_cache( (char *) &g_execute_here[index] );
      ++index;
    }

  g_execute_here[index++] = g_dummy_retw_ptr[0];
  _flush_i_cache( (char *)&g_execute_here[index] );

  g_execute_here[index++] = g_dummy_retw_ptr[1];
  _flush_i_cache( (char *)&g_execute_here[index] );

  g_execute_here[index++] = g_dummy_retw_ptr[2];
  _flush_i_cache( (char *)&g_execute_here[index] );


  _xmon_execute_here(a4_value, &g_execute_here[0]);

  success = 1;

exit_gracefully:

  _xmon_set_cpenable( old_cpe );

  return success;
}




static void
_tell_gdb ( int sigval )
{
  int tt;                       /* Trap type */
  int addr;
  int length;
  int value;
  int intlevel;
  int woe;
  char *ptr;
  unsigned long *sp;
  unsigned int wb, pc;
  int i;
  struct hw_break_info *bp;

  wb = SR_REG(WINDOWBASE);
  sp = (unsigned long *) reg_at_wb( SP_REGNUM, wb );
  pc = SR_REG(PC);
  intlevel = SR_REG(DBG_EPS) & 0x0f;
  woe      = SR_REG(DBG_EPS) & 0x040000;

  /* If we find that window overflow/underflow enabled and
     the interrupt level zero then we can safely save all
     the registers to the stack.
  */
```

```
  if( woe != 0 && intlevel==0 ) {
    if(save_to_stack()!=0) {
      ptr = remcomOutBuffer;
      *ptr++ = 'E';
      if( error_log_end != 0 && error_log_end < &error_log[LOG_SIZE-1]) {
        error_log_end[0] = 0;
        _strcpy( ptr, error_log );
        mon_error_clear();
      } else
        _strcpy( ptr, "Error in save_to_stack\n" );
      putpacket(remcomOutBuffer);
    }
  }

  ptr = remcomOutBuffer;

  /* Tell gdb that we have stopped */
  *ptr++ = 'S';
  *ptr++ = hexchars[sigval >> 4];
  *ptr++ = hexchars[sigval & 0xf];
  *ptr++ = 0;
  putpacket(remcomOutBuffer);

  while (1)
    {
      remcomOutBuffer[0] = 0;

      getpacket(remcomInBuffer);
      switch (remcomInBuffer[0])
        {
        case '?':
          remcomOutBuffer[0] = 'S';
          remcomOutBuffer[1] = hexchars[sigval >> 4];
          remcomOutBuffer[2] = hexchars[sigval & 0xf];
          remcomOutBuffer[3] = 0;
          break;

        case 'E': /* xtensa specific */
          /* send the error message log back */
          /*  mem2hex( error_log, remcomOutBuffer, error_log_end-error_log, 0 );*/
          break;

        case 'd':
                              /* toggle debug flag */
          break;

#if 0
        case 'g':             /* return the value of the CPU registers */
          {
            ptr = mem2hex( (char *)_registers, remcomOutBuffer,
                           SAVE_AREA_SIZE, 0 );
          }
          break;

        case 'G':         /* set the value of the CPU registers - return OK */
          {
            /* We allow the user to set any registers without checking.
               Users can wedge the board if they load inconsistent values
               into the registers */
            write_mem( &remcomInBuffer[1], (char *)_registers,
                       SAVE_AREA_SIZE, 0, 0 );
            aok();
          }
          break;
#endif

        case 'p':
          ptr = &remcomInBuffer[1];
          if( hexToInt(&ptr, &addr) )
            {
              unsigned char *reg_ptr = NULL;
```

```
        // !!@
        //if(mem2hex( (char *)&_registers[addr], remcomOutBuffer, 4, 0 ))
        //       break;
        reg_ptr = get_reg_ptr( addr);
        if (reg_ptr != NULL)
          {
            if(mem2hex( reg_ptr, remcomOutBuffer, 4, 0 ))
              break;
          }
        _strcpy( remcomOutBuffer, "Error: read failure" );
      }
    else
      bad_protocol();
    break;

  case 'P':
    ptr = &remcomInBuffer[1];
    if( hexToInt(&ptr, &addr) && *ptr++ == '='
        && write_mem( ptr, (char *)&value, 4, 0, 0 ))
      {
        unsigned char *reg_ptr = NULL;
        if (set_reg_value(addr, value))
          {
            aok();
          }
        else
          {
            _strcpy( remcomOutBuffer, "Error: write failure" );
          }
        // !!@
        //              _registers[addr] = value;
      }
    else
      bad_protocol();
    break;

  case 'm':        /* mAA..AA,LLLL  Read LLLL bytes at address AA..AA */
    /* Try to read %x,%x.  */

    ptr = &remcomInBuffer[1];

    if (hexToInt(&ptr, &addr)
        && *ptr++ == ','
        && hexToInt(&ptr, &length))
      {
        if (mem2hex((char *)addr, remcomOutBuffer, length, 1))
          break;
        _strcpy( remcomOutBuffer, "Error: read failure" );
      }
    else
      bad_protocol();
    break;

  case 'M': /* MAA..AA,LLLL: Write LLLL bytes at address AA.AA return OK */
    /* Try to read '%x,%x:'.  */

    ptr = &remcomInBuffer[1];

    if (hexToInt(&ptr, &addr)
        && *ptr++ == ','
        && hexToInt(&ptr, &length)
        && *ptr++ == ':')
      {
        if (write_mem(ptr, (char *)addr, length, 1, 1))
          aok();
        else
          _strcpy(remcomOutBuffer, "Error: write failure");
      }
    else
      bad_protocol();
```

```
        break;

      case 'c':    /* cAA..AA    Continue at address AA..AA(optional) */
        /* try to read optional parameter, pc unchanged if no parm */

        ptr = &remcomInBuffer[1];
        if (hexToInt(&ptr, &addr))
          {
            SR_REG(PC) = addr;
          }
        //      else
        //        bad_protocol();
        return;

      case 's':
          /* use icount mechanism to step a single instruction */
          set_icount_for_single_step(intlevel);
          return;

        /* kill the program */
      case 'k' :            /* do nothing */
        break;

        /* Xtensa specific commands */
      case 'X':
        switch( remcomInBuffer[1] )
          {
          case 'q':
            switch (remcomInBuffer[2])
              {
              case 'n':
                _strcpy( remcomOutBuffer, "XMON2.5" );
                break;

              case 'p':
                _strcpy( remcomOutBuffer, "n");
                break;

              case 'P':
                _strcpy( remcomOutBuffer, "n");
                break;

              default:
                break;
              }
            break;

          case 'e':
            if ( remcomInBuffer[2] == 'x' && remcomInBuffer[3] == 'e' )
              {
                ExecuteSomeInstruction( &remcomInBuffer[4] );
                remcomOutBuffer[0] = '\0';
              }
            break;


          case 'B':
            /* Set a breakpoint using the ibreak registers */
#if NIBREAK==0
            _strcpy( remcomOutBuffer, "Error: configuration has no IBREAK registers");
#else
            ptr = &remcomInBuffer[4];
            if( !hexToInt(&ptr, &addr ) ) {
              bad_protocol();
              break;
            }


            switch( remcomInBuffer[2] ) {
            case 's':                /* set */
              for( i = 0, bp = hw_break; i < NIBREAK; i++, bp++ )
```

```
                if( bp->free ) {
                  bp->free = 0;
                  bp->addr = (char *)addr;
                  // !!@
                  //_registers[bp->reg_number] = addr;
                  SR_REG(IBREAKENABLE) |= (1<<i);
                  aok();
                  break;
                }
              if( i >= NIBREAK )
                _strcpy( remcomOutBuffer, "Error: out of ibreak registers");
              break;
            case 'r':           /* remove */
              for( i = 0, bp = hw_break; i < NIBREAK; i++, bp++ )
                if( !bp->free && bp->addr == (char *)addr ) {
                  bp->free = 1;
                  bp->addr = 0;
                  // !!@
                  //_registers[bp->reg_number] = 0;
                  SR_REG(IBREAKENABLE) &= ~(1<<i);
                  aok();
                  break;
                }
              if( i >= NIBREAK )
                _strcpy( remcomOutBuffer, "Error: breakpoint not found");
              break;
            default:
              break;
            }
#endif
          }
        break;
#if 0
      case 't':             /* Test feature */
        asm (" std %f30,[%sp]");
        break;
      case 'r':             /* Reset */
        asm ("call 0
              nop ");
        break;
#endif

#if 0
Disabled until we can unscrew this properly

      case 'b':       /* bBB...  Set baud rate to BB... */
        {
          int baudrate;
          extern void set_timer_3();

          ptr = &remcomInBuffer[1];
          if (!hexToInt(&ptr, &baudrate))
            {
              _strcpy(remcomOutBuffer,"B01");
              break;
            }

          /* Convert baud rate to uart clock divider */
          switch (baudrate)
            {
            case 38400:
              baudrate = 16;
              break;
            case 19200:
              baudrate = 33;
              break;
            case 9600:
              baudrate = 65;
              break;
            default:
              _strcpy(remcomOutBuffer,"B02");
```

```
                goto x1;
              }

            putpacket("OK");   /* Ack before changing speed */
            set_timer_3(baudrate); /* Set it */
          }
x1:       break;
#endif
        }                       /* switch */

      /* reply to the request */
      putpacket(remcomOutBuffer);
    }
}
```

```
/*************************************************************

Xtensa hardware-dependent utilities

*************************************************************/

/* retrieved register at a particular window base */
static long reg_at_wb( unsigned int reg, unsigned int wb )
{
  unsigned int relocated_reg;
  if( (NUM_AREGS/4) <= wb )
    mon_error( "invalid window base in reg_at_wb\n" );
  if( NUM_VISIBLE_AREGS <= reg )
    mon_error( "invalid register in reg_at_wb\n" );
  relocated_reg = (reg + (wb<<2)) & AREGS_MASK;
  //  relocated_reg += AR0_OFFSET/4;

  return _ar_registers[relocated_reg];
}

/* retrieved register in window of interrupt process */
static int reg_at_ipwb( unsigned int reg )
{
  return reg_at_wb( reg, SR_REG(WINDOWBASE) );
}

static int save_to_stack()
{
  int i;
  int ws = SR_REG(WINDOWSTART);
  int wb = SR_REG(WINDOWBASE);
  int callee_win, win;
  long *sp, *caller_sp;

  /* rotate so that first bit of ws corresponds to
     wb+1 */
  ws = (ws >> (wb+1)) | ( ws << (NUM_AREGS/4-(wb+1)) );
  ws &= WS_MASK;

  /* find first window after ipwb */
  if( ws == 0 )
    mon_error( "window start zero in save_to_stack\n" );
  for( i = 0; (ws&1)==0; i++ )
    ws >>= 1;
  ws >>= 1;
  i++;
  while( ws != 0 )
    {
      win = (wb+i) & WB_MASK;
      if( ws & 1 )
        {
          callee_win = (win+1) & WB_MASK;
          sp    = (long *)reg_at_wb( 1, callee_win) - 4;
          sp[0] = reg_at_wb( 0, win );
          sp[1] = reg_at_wb( 1, win );
          sp[2] = reg_at_wb( 2, win );
          sp[3] = reg_at_wb( 3, win );
          i = i+1;
          ws >>= 1;
          continue;
        }
      if( ws & 2 )
        {
          callee_win = (win+2) & WB_MASK;
          sp    = (long *)reg_at_wb( 1, callee_win) - 4;
          sp[0] = reg_at_wb( 0, win );
          sp[1] = (long) caller_sp = (long *)reg_at_wb( 1, win );
          sp[2] = reg_at_wb( 2, win );
          sp[3] = reg_at_wb( 3, win );
```

```
            /* save a4 thru a7 */
            caller_sp = (long *)caller_sp[-3];
            caller_sp[-8] = reg_at_wb( 4, win );
            caller_sp[-7] = reg_at_wb( 5, win );
            caller_sp[-6] = reg_at_wb( 6, win );
            caller_sp[-5] = reg_at_wb( 7, win );
            i = i+2;
            ws >>= 2;
            continue;
          }
        if( ws & 4 )
          {
            callee_win = (win+2) & WB_MASK;
            sp    = (long *)reg_at_wb( 1, callee_win) - 4;
            sp[0] = reg_at_wb( 0, win );
            sp[1] = (long) caller_sp = (long *)reg_at_wb( 1, win );
            sp[2] = reg_at_wb( 2, win );
            sp[3] = reg_at_wb( 3, win );

            /* save a4 thru a11 */
            caller_sp = (long *)caller_sp[-3];
            caller_sp[-12] = reg_at_wb(  4, win );
            caller_sp[-11] = reg_at_wb(  5, win );
            caller_sp[-10] = reg_at_wb(  6, win );
            caller_sp[ -9] = reg_at_wb(  7, win );
            caller_sp[ -8] = reg_at_wb(  8, win );
            caller_sp[ -7] = reg_at_wb(  9, win );
            caller_sp[ -6] = reg_at_wb( 10, win );
            caller_sp[ -5] = reg_at_wb( 11, win );
            i = i+3;
            ws >>= 3;
            continue;
          }

        /* ERROR Condition: illegal window size,
           return an error indication and message */
        mon_error("illegal window size\n" );
        return -1;
      }
    return 0;
}

void putDebugString(char *s)
{
  while(*s)
    {
      putDebugChar(*s);
      s++;
    }
}

static void putDebugChar(char c)
{
  if( _in_simulator )
    _xmon_out(c);
  else {
    _uart_out((uart_dev_t *)XT1000_DUART_1_ADDR, c );
  }
}

static int getDebugChar()
{
  return _in_simulator ? _xmon_in() : _uart_in((uart_dev_t *)XT1000_DUART_1_ADDR);
}

static int flushDebug()
{
  return _in_simulator ? _xmon_flush() : 0;
}
```

```
static int fetchUserRegister()
{
  __asm__("nop");
  return;
}

static void setUserRegister()
{
  __asm__("nop");

  return;
}
```